CSE503: Software Engineering

David Notkin
University of Washington
Department of Computer Science & Engineering
Winter 2002

Information hiding

- Information hiding is perhaps the most important intellectual tool developed to support software design [Parnas 1972]
  – Makes the anticipation of change a centerpiece in decomposition into modules
- Provides the fundamental motivation for abstract data type (ADT) languages
  – And thus a key idea in the OO world, too

Basics of information hiding

- Modularize based on anticipated change
  – Fundamentally different from Brooks' approach in OS/360 (see old and new MMM)
- Separate interfaces from implementations
  – Implementations capture decisions likely to change
  – Interfaces capture decisions unlikely to change
  – Clients know only interface, not implementation
  – Implementations know only interface, not clients
- Modules are also work assignments

Anticipated changes

- The most common anticipated change is "change of representation"
  – Anticipating changing the representation of data and associated functions (or just functions)
  – Again, a key notion behind abstract data types
- Ex:
  – Cartesian vs. polar coordinates; stacks as linked lists vs. arrays; packed vs. unpacked strings

Claim

- We less frequently change representations than we used to
  – We have significantly more knowledge about data structure design than we did 25 years ago
  – Memory is less often a problem than it was previously, since it's much less expensive
- Therefore, we should think twice about always anticipating that representations will change
  – This is important, since we can't simultaneously anticipate all changes
  – Ex: Changing the representation of null-terminated strings in Unix systems wouldn't be sensible
    - And this doesn't represent a stupid design decision

Other anticipated changes?

- Information hiding isn't only ADTs
- Algorithmic changes
  – (These are almost always part and parcel of ADT-based decompositions)
  – Monolithic to incremental algorithms
  – Improvements in algorithms
- Replacement of hardware sensors
  – Ex: better altitude sensors
- More?

Ubiquitous computing domain

- Portolano is a UW CSE project on this topic
  - Devices everywhere, handhelds, on-body devices, automated laboratories, etc.
- The set of anticipated changes is significantly different than in many other domains
  - Data is more stable than computations
  - Must accommodate diversity in communication speed, reliability, etc.
- Interesting domain for information hiding

Central premise I

- We can effectively anticipate changes
  - Unanticipated changes require changes to interfaces or (more commonly) simultaneous changes to multiple modules
- How accurate is this premise?
  - We have no idea
  - There is essentially no research about whether anticipated changes happen
  - Nor do we have disciplined ways to figure out how to better anticipate changes

The A-7 Project

- In the late 1970's, Parnas led a project to redesign the software for the A-7 flight program
  - One key aspect was the use of information hiding
- The project had successes, including a much improved specification of the system and the definition of the SCR requirements language
- But little data about actual changes was gathered

Central premise II

- Changing an implementation is the best change, since it's isolated
- This may not always be true
  - Changing a local implementation may not be easy
  - Some global changes are straightforward
    - Mechanically or systematically
  - VanHilst's work showed an alternative
    - Using parameterized classes with a deferred supertype [ISOTAS, FSE, OOPSLA]
  - Griswold's work on information transparency

Central premise III

- The semantics of the module must remain unchanged when implementations are replaced
  - Specifically, the client should not care how the interface is implemented by the module
- But what captures the semantics of the module?
  - The signature of the interface? Performance? What else?

Central premise IV

- One implementation can satisfy multiple clients
  - Different clients of the same interface that need different implementations would be counter to the principle of information hiding
    - Clients should not care about implementations, as long as they satisfy the interface
  - Kiczales' work on open implementations

Information hiding reprise

- It's probably the most important design technique we know
- And it's broadly useful
- It raised consciousness about change
- But one needs to evaluate the premises in specific situations to determine the actual benefits (well, the actual potential benefits)

Information Hiding and OO

- Are these the same? Not really
  - – OO classes are chosen based on the domain of the problem (in most OO analysis approaches)
  - – Not necessarily based on change
- But they are obviously related (separating interface from implementation, e.g.)
- What is the relationship between sub- and super-classes?

Layering [Parnas 79]

- A focus on information hiding modules isn't enough
- One may also consider abstract machines
  - – In support of program families
    - Systems that have "so much in common that it pays to study their common aspects before looking at the aspects that differentiate them"
- Still focusing on anticipated change

The `uses` relation

- A program `A` uses a program `B` if the correctness of `A` depends on the presence of a correct version of `B`
- Requires specification and implementation of `A` and the specification of `B`
- Again, what is the "specification"? The interface? Implied or informal semantics?
  - – Can `uses` be mechanically computed?

uses vs. invokes

- These relations often but do not always coincide
- Invocation without use: name service with cached hints
- Use without invocation: examples?

```
ipAddr := cache(hostName);
if wrong(ipAddr,hostName) then
   ipAddr := lookup(hostName)
endif
```

Parnas' observation

- A non-hierarchical uses relation makes it difficult to produce useful subsets of a system
  - – It also makes testing difficult
  - – (What about upcalls?)
- So, it is important to design the `uses` relation

Criteria for `uses(A,B)`

- `A` is essentially simpler because it uses `B`
- `B` is not substantially more complex because it does not use `A`
- There is a useful subset containing `B` but not `A`
- There is no useful subset containing `A` but not `B`

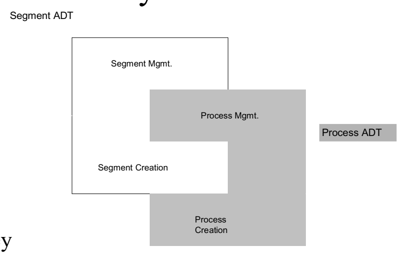
Layering in THE
(Dijkstra’s layered OS)

- OK, those of you who took OS
- How was layering used, and how does it relate to this work?

- (For thinking about off-line, or for email discussion)

Modules and layers interact?

- Information hiding modules and layers are distinct concepts
- How and where do they overlap in a system?



Language support

- We have lots of language support for information hiding modules
  - C++ classes, Ada packages, etc.
- We have essentially no language support for layering
  - Operating systems provide support, primarily for reasons of protection, not abstraction
  - Big performance cost to pay for “just” abstraction

Next lecture

- Implicit invocation
  - Essentially, event-based design